Practice Final Exam

CSCI-400: Principles of Programming Languages (all sections)

Instructor: Dr. Neil T. Dantam

Spring 2024

SOLUTIONS

Duration: 120 minutes

Instructions:

1. Please silence or turn off any phones, tablets, or electronic devices.
2. Write your name on EVERY page, BOTH SIDES.
3. This exam is closed book.
4. You may use one letter-sized sheet of notes, both-sides.
5. Answers must be NEAT and LEGIBLE. Credit cannot be given for illegible answers.
6. Exams will be scanned:
    (a) Write only within the marked borders.
    (b) Do not staple or dog-ear pages.
7. For questions asking for proofs:
    (a) Proofs must be precise and mathematically justified.
    (b) Good proofs are concise. Include sufficient detail, but do not include irrelevant information.
    (c) Please do not transcribe algorithms or proofs covered in lecture onto the exam. You may just reference these results as lemmas or subroutines for full credit (if used appropriately).
8. For questions asking for code or pseudocode: you may use OCaml syntax or pseudocode of the style used in lecture.

1. Consider a language based on the lambda calculus that includes function definition, function application, numbers, and arithmetic operations (addition, subtraction, multiplication, division):

   (a) Write a context-free grammar for this language.

$$\begin{array}{lcl} \langle e \rangle & \rightarrow & \text{“number”} \mid \text{“var”} \\ & \mid & \text{“}\lambda\text{” “var” “.”} \langle e \rangle \\ & \mid & \langle e \rangle \langle e \rangle \\ & \mid & \langle e \rangle \langle b \rangle \langle e \rangle \\ & \mid & \text{“(”} \langle e \rangle \text{“)”} \\ \langle b \rangle & \rightarrow & \text{“+”} \mid \text{“-”} \mid \text{“*”} \mid \text{“/”} \end{array}$$

   (b) Write a variant for the abstract syntax of this language.

```
type bop ←
  | Add
  | Sub
  | Mul
  | Div
type expr ←
  | Num of number
  | Var of string
  | FuncExpr of string × expr
  | CallExpr of expr × expr
  | BopExpr of expr × bop × expr
```

(c) Formally define the big step semantics of this language.

- $\frac{v \in \mathbb{Z}}{v \Downarrow v}$ Num
- $\frac{\cdot}{E \vdash x \Downarrow E(x)}$ Var
- $\frac{\cdot}{E \vdash \boldsymbol{\lambda} x\,.\,t \Downarrow (\boldsymbol{\lambda} x\,.\,t,\, E)}$ Func
- $\frac{E_1 \vdash t_1 \Downarrow (\boldsymbol{\lambda} x\,.\,t_3,\, E_2) \quad E_1 \vdash t_2 \Downarrow v_1 \quad E_2[x \mapsto v_1] \vdash t_3 \Downarrow v_2}{E_1 \vdash t_1\ t_2 \Downarrow v_2}$ Call
- $\frac{E \vdash t_1 \Downarrow v_1 \quad E \vdash t_2 \Downarrow v_2}{E \vdash t_1 + t_2 \Downarrow v_1 + v_2}$ Add
- $\frac{E \vdash t_1 \Downarrow v_1 \quad E \vdash t_2 \Downarrow v_2}{E \vdash t_1 - t_2 \Downarrow v_1 - v_2}$ Sub
- $\frac{E \vdash t_1 \Downarrow v_1 \quad E \vdash t_2 \Downarrow v_2}{E \vdash t_1 * t_2 \Downarrow v_1 * v_2}$ Mul
- $\frac{E \vdash t_1 \Downarrow v_1 \quad E \vdash t_2 \Downarrow v_2}{E \vdash t_1 / t_2 \Downarrow v_1 / v_2}$ Div

(d) Write an evaluator for this language based on your big step semantics.

```
defun eval $E$ $t$ →
  match $t$ with
    case Num($n$) → $n$
    case Var($x$) → $E(x)$
    case FuncExpr($v$, $t$) → (FuncExpr($v$, $t$), $E$)
    case BopExpr($x$, $b$, $y$) →
      let
        $x$ ← eval $E$ $x$
        $y$ ← eval $E$ $y$
      in
        match $b$ with
          case Add → x+y
          case Sub → x-y
          case Mul → x*y
          case Div → x/y
    case CallExpr($f$, $a$) →
      match (eval $E$ $f$, eval $E$ $a$) with
        case ((FuncExpr($x$, $t$), $E$), $v$) →
          eval $E[x \mapsto v]$ $t$
        otherwise ERROR
```

2. Using your language syntax from Question 1:
   (a) Write an expression containing at least two function definitions and two function applications.
   (b) Evaluate this expression using *dynamic* scope.
   (c) Evaluate this expression using *lexical* scope.

   Varies

3. Using your language from Question 1:

   (a) Write an expression containing at least one function definition, one function application, and one arithmetic operation. Then, use the basic constraint-generation/solving approach discussed in class to infer the type of your expression.

   (b) What are the preliminary types?

   (c) What are the constraints?

   (d) What is the solution to the constraints obtained from unification?

   Varies

4. For the following data type and functions, use structural induction to prove that for any list with at least one element, `listor`(ℓ) must be true whenever `listand`(ℓ) is true.

```
type boollist ←
    | Nil
    | Cons of $\mathbb{B} \times$ boollist
```

```
Function listand($\ell$)
match $\ell$ with
    case nil → true
    case cons false $\ell$ →  false
    case cons true $\ell$ →
        listand($\ell$)
```

```
Function listor($\ell$)
match $\ell$ with
    case nil → false
    case cons true $\ell$ →  true
    case cons false $\ell$ →
        listor($\ell$)
```

Proof. We prove by induction that $\texttt{listand}(\ell) \implies \texttt{listor}(\ell)$ for $|\ell| \geq 1$.

- Basis: $\texttt{listor}([\textbf{true}]) = \texttt{listand}([\textbf{true}]) = \textbf{true}$
- Induction:
  - Assume that for list ℓ, $\texttt{listor}(\ell) = \texttt{listand}(\ell) = \textbf{true}$
  - There are two cases to consider: $\ell' = \texttt{cons}\,(\textbf{false},\ \ell)$ and $\ell'' = \texttt{cons}\,(\textbf{true},\ \ell)$
  - $\texttt{listand}(\ell') = \textbf{false}$, so the property (implication) offers no information about the value of $\texttt{listor}(\ell')$. That is, $\Big(\texttt{listand}(\ell') \implies \texttt{listor}(\ell')\Big) \rightsquigarrow \Big(\textbf{false} \implies \texttt{listor}(\ell')\Big) \rightsquigarrow \textbf{true}$
  - $\texttt{listand}(\ell'') = \texttt{listor}(\ell'') = \textbf{true}$.
  - Inductively, the property must hold.

□